# Toplevel like tool and documentation for Benjamin C. Pierce's untyped lambda calculus library

## Introduction

The task in hand was to develop a small toplevel tool, a REPL environment, where a lambda-calculus expression can be tested. The tool is make use of the untyped lambda implementation available at Benjamin C. Pierce's site[1]. Some comments were added to the original source code using Ocamldoc.

## Obtaining and building

This work is available as a Github project, can be cloned on any Linux or Unix environment with:

```
git clone https://github.com/diegots/fic-dlp-lambda.git lambda-toplevel
```

A directory called `lambda-toplevel` will be created with all necessary source files. The original `Makefile` was modified to include new targets. To use it just go inside `lambda-toplevel` and use make tool as usual:

```
cd lambda-toplevel
make toplevel
```

## Basic usage

Once compiled, using the tool is straightforward, just calling the toplevel binary like any other binary will work:

```
diego@CompaqCQ57:fic-dlp-lambda$ ./toplevel
Welcome, this is a lambda-calculus toplevel interpreter based on Benjamin
C. Pierce implementations available at
https://www.cis.upenn.edu/~bcpierce/tapl/

>>
```

Developed application is compatible with readline utilities like `rlwrap`. It's an easy way to have line editing and persistent history.

Finish the session can be done with `exit` command or using `Crtl-D`. Of course this toplevel remembers previous declarations so a variable can be named anytime after bounding.

1 https://www.cis.upenn.edu/~bcpierce/tapl/checkers/fulluntyped.tar.gz

Now some examples are presented. Of course, all examples from `test-examples` included in the original distribution can be tried out. First, to bind a symbol and use it later, the slash operator can be used:

```
>> x/;
x
>> x;
x
>>
```

Declaring an integer value and testing if it zero:

```
>> nz = 2;
nz = 2
>> iszero nz;
false
>> succ nz;
3
>>
```

Getting the predecessor of a value:

```
>> y = lambda j.(pred j);
y = lambda j. pred j
>> y 5;
4
>>
```

Declaring a record with mixed values:

```
>> {b=1, 3, "hello", onedotthree=1.3};
{b=1, 3, "hello", onedotthree=1.3}
>>
```

Infinite loop:

```
Infinite loop example
p = lambda x'. x' x'
>> p;
(lambda x'. x' x')
>> p p;
Fatal error: exception Stack_overflow
diego@CompaqCQ57:fic-dlp-lambda$
```

Accessing the third element from record:

```
>> {a=3, b="hello", c=2.3}.c;
2.3
```

Functions to get first or second element from a pair:

```
>> fst = lambda a. lambda b. a;
fst = lambda a. lambda b. a
>> snd = lambda a. lambda b. b;
snd = lambda a. lambda b. b
>>
>> fst 2 3;
2
>> snd 2 3;
3
>>
```

Square function for float point numbers:

```
>> square = lambda x. timesfloat x x;
square = lambda x. timesfloat x x
>> square 9.0;
81.
>>
```

## Generating documentation

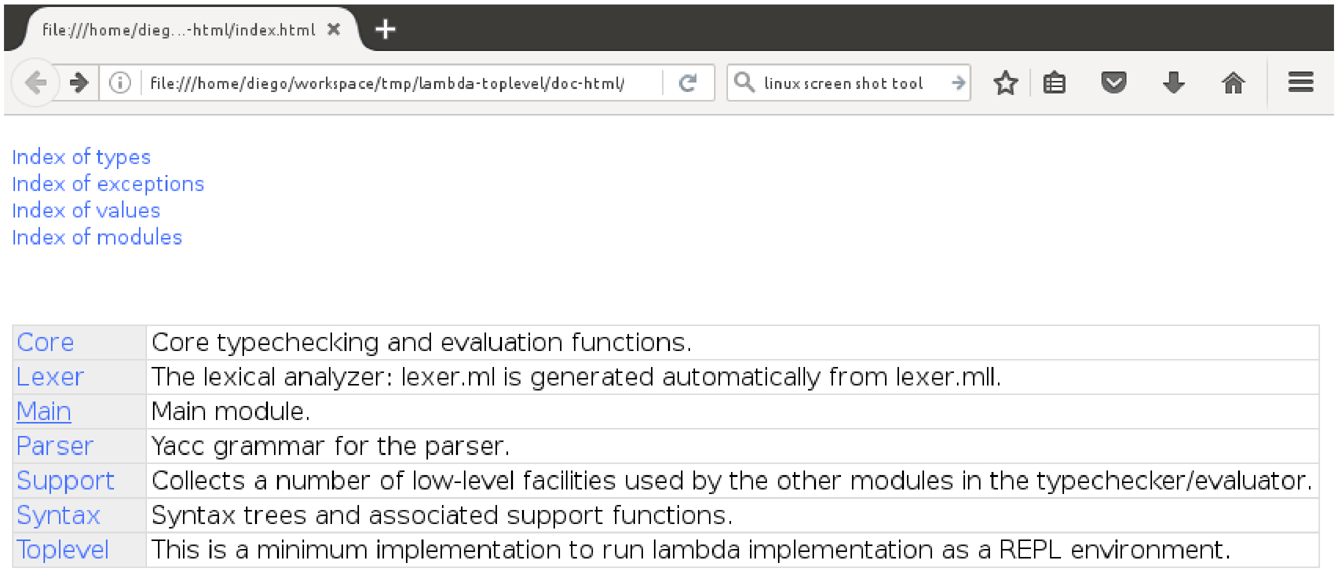
Source code was documented and can be read directly with any editor but `Ocamldoc` was used and a nice HTML reference can be generated with the following instruction:

```
make doc
```

And accessed with any browser:

```
firefox doc-html/index.html &
```

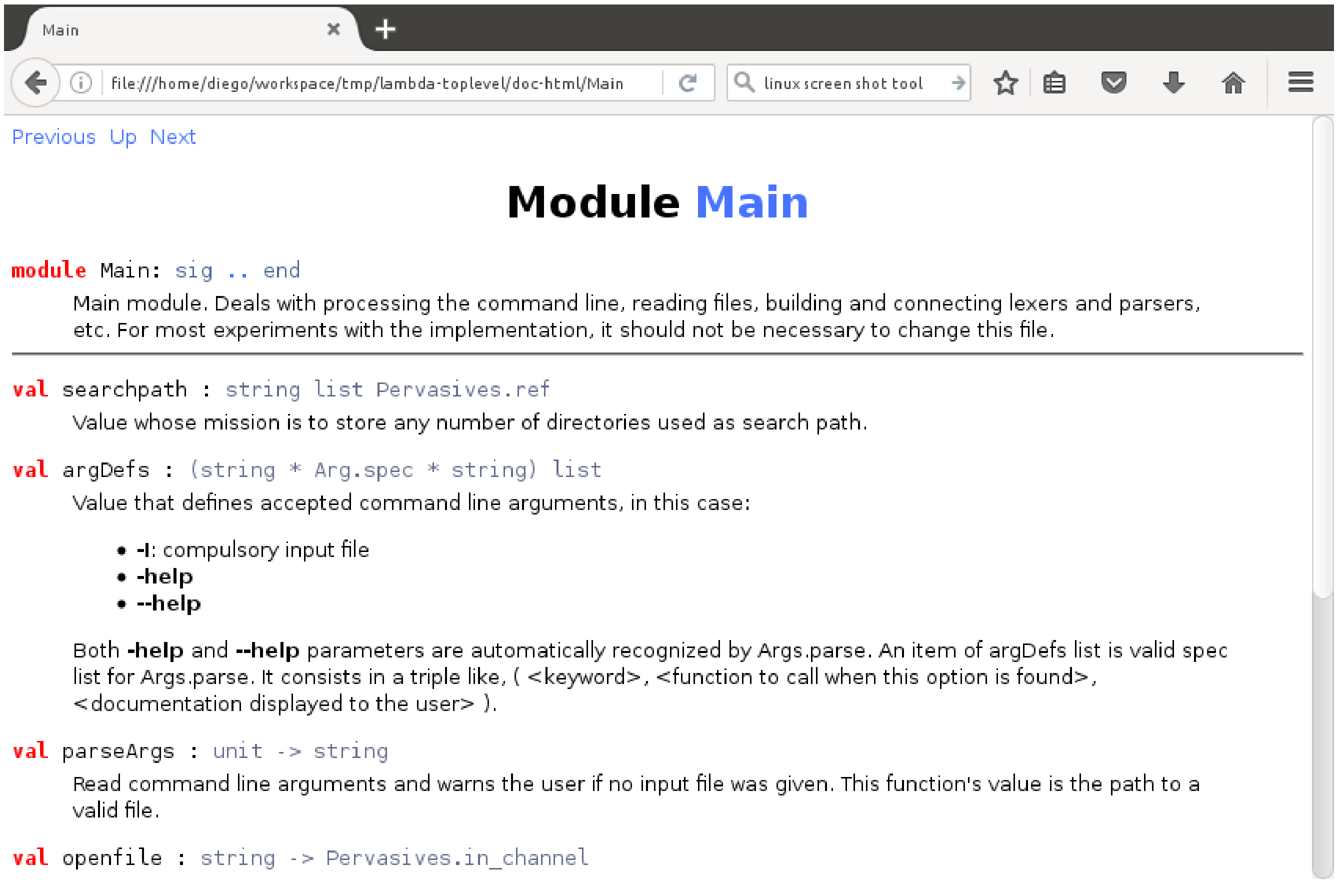
This will open the main reference page shown on Illustration 1. From this page any module can be accessed, like the Main module shown on the Illustration 2.


file:///home/dieg...-html/index.html
file:///home/diego/workspace/tmp/lambda-toplevel/doc-html/
linux screen shot tool

Index of types
Index of exceptions
Index of values
Index of modules

| | |
|---|---|
| Core | Core typechecking and evaluation functions. |
| Lexer | The lexical analyzer: lexer.ml is generated automatically from lexer.mll. |
| Main | Main module. |
| Parser | Yacc grammar for the parser. |
| Support | Collects a number of low-level facilities used by the other modules in the typechecker/evaluator. |
| Syntax | Syntax trees and associated support functions. |
| Toplevel | This is a minimum implementation to run lambda implementation as a REPL environment. |

*Illustration 1: Ocamldoc reference main view*


Main
file:///home/diego/workspace/tmp/lambda-toplevel/doc-html/Main
linux screen shot tool
Previous Up Next

Module Main

```
module Main: sig .. end
```

Main module. Deals with processing the command line, reading files, building and connecting lexers and parsers, etc. For most experiments with the implementation, it should not be necessary to change this file.

```
val searchpath : string list Pervasives.ref
```

Value whose mission is to store any number of directories used as search path.

```
val argDefs : (string * Arg.spec * string) list
```

Value that defines accepted command line arguments, in this case:

- **-I**: compulsory input file
- **-help**
- **--help**

Both **-help** and **--help** parameters are automatically recognized by Args.parse. An item of argDefs list is valid spec list for Args.parse. It consists in a triple like, ( <keyword>, <function to call when this option is found>, <documentation displayed to the user> ).

```
val parseArgs : unit -> string
```

Read command line arguments and warns the user if no input file was given. This function's value is the path to a valid file.

```
val openfile : string -> Pervasives.in_channel
```

*Illustration 2: Ocamldoc main module detail*

A high-level graph of module dependencies can be also generated thanks to Ocamldoc. It's a Dot[2] graph but Makefile exports it as a svg file of name `modules-graph.svg`. Write the following instruction to obtain it:

```
make doc-graph
```

2 http://www.graphviz.org/Documentation.php

## What changed from the original source code

The starting idea was to avoid editing the original source files as much as possible. With that in mind, the workflow involved to read the source code to understand it and comment it out. Then, a point was reached when gained knowledge was enough to intercept the right function calls to skip opening a file to parse it but write a loop and pass the lexical analyzer standard input lines. There was a twist because first versions didn't keep record of what lambda terms were already been defined. Hopefully original authors thought also about this problem and created a memory called *context* or *ctx* in the source files. Those are all the necessary pieces that finally become the REPL environment. Original Makefile was modified to include all new targets mentioned above: *toplevel*, *doc*, *doc-graph*.

Bellow are two interaction diagrams that give an idea of the calls order and the difference between *main.ml* and *toplevel.ml*. It's not exhaustive and maybe not the strictly right way of doing this class of diagram, given that this it no Object Oriented code but helps to clarify things a bit.

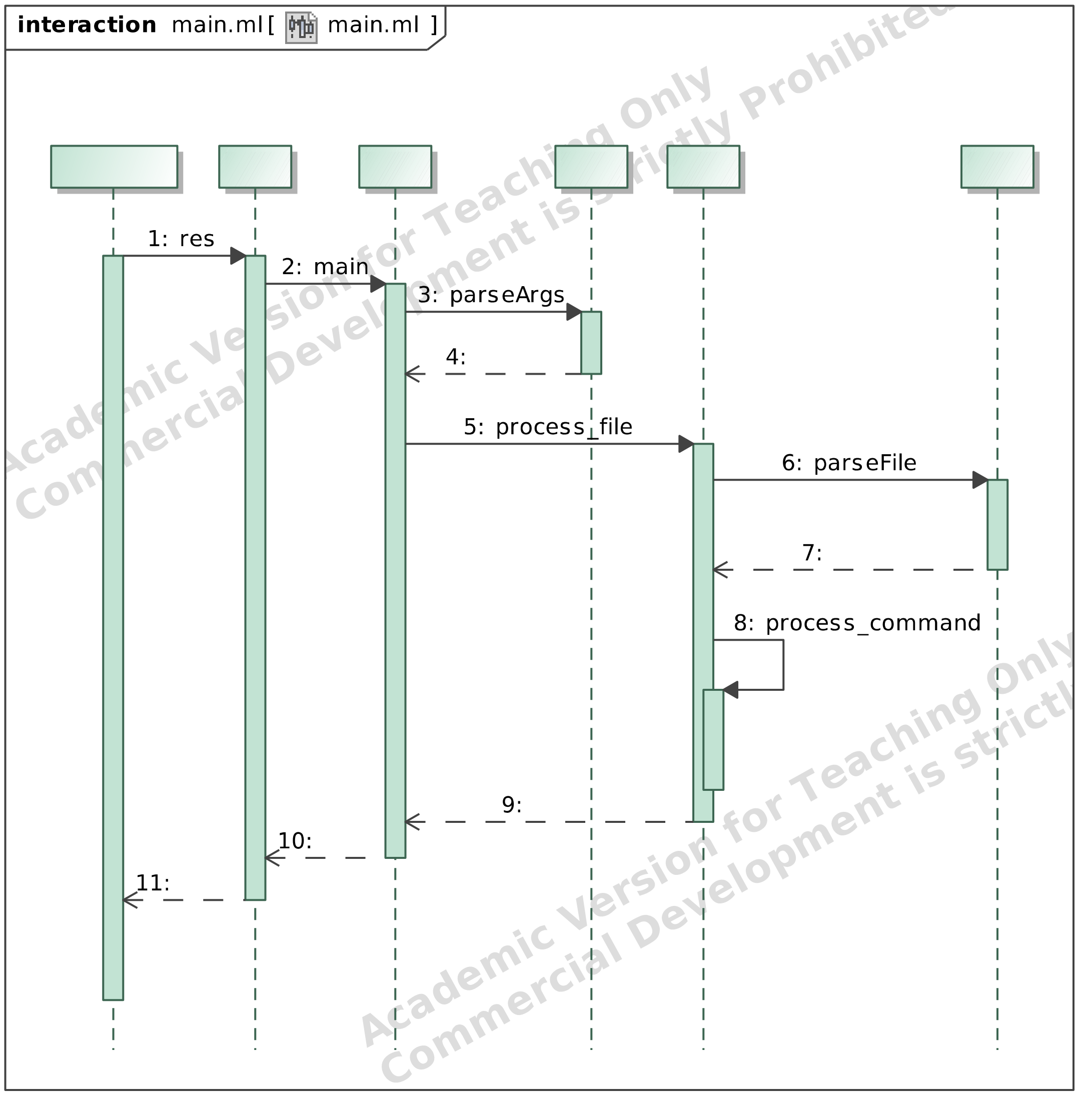


*Illustration 3: Original code calls*

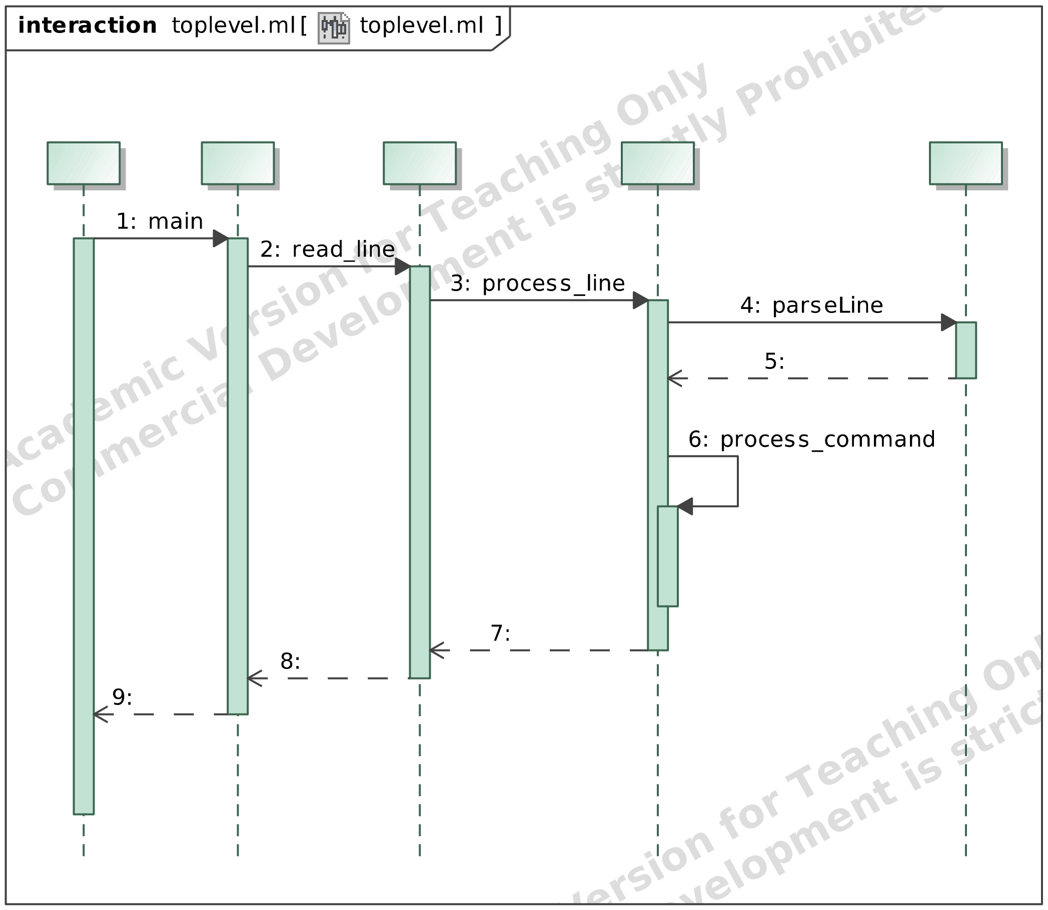


*Illustration 4: Developed code calls*